Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Computação e Tecnologia – DCT
Bacharelado em Sistemas de Informação – BSI

Comparando Estruturas de Dados: Análise de Desempenho de Árvores Binárias, Árvores AVL e Tabela-Hash

Charles Eduardo Araújo de Faria

Orientador: Prof. Dr. João Paulo de Souza Medeiros

Relatório Técnico apresentado ao Curso de Bacharelado em Sistemas de Informação como parte dos requisitos para obtenção de nota na disciplina de Estrutura de Dados.

Caicó, RN, 30 de junho de 2023

Resumo

Este trabalho consiste no desenvolvimento de uma atividade prática que aborda a análise de algoritmos relacionados ao problema de busca em três estruturas de dados: árvore binária, árvore balanceada (AVL) e tabela de dispersão (Hash). O objetivo é explorar e comparar as características e eficiência dessas estruturas no contexto da busca. A atividade proporcionará uma compreensão mais aprofundada dos algoritmos e estruturas de dados envolvidos, permitindo uma análise de suas vantagens e desvantagens.

Palavras-chave: Análise de algoritmos; Busca; Desempenho; Árvore binária; Árvore balanceada (AVL); Tabela de dispersão (Hash).

Abstract

This work consists of the development of a practical activity that addresses the analysis of algorithms related to the search problem in three data structures: binary tree, balanced tree (AVL) and hash table (Hash). The objective is to explore and compare the characteristics and efficiency of these structures in the search context. The activity will provide a deeper understanding of the algorithms and data structures involved, allowing an analysis of their advantages and disadvantages.

Keywords: Analysis of algorithms; Search; Performance; Binary tree; Balanced tree (AVL); Hash table.

Sumário

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

1. Introdução

As árvores binárias e AVL, juntamente com as tabelas-hash, são estruturas de dados específicas que utilizam algoritmos para organizar e manipular dados de acordo com regras definidas. Esses algoritmos, assim como os algoritmos de ordenação, são cruciais para a programação moderna. Suas implementações serviram de base para o desenvolvimento de estruturas mais complexas, frequentemente utilizadas no desenvolvimento de software.

Em resumo, o objetivo em comum da árvore binária, árvore AVL e tabela-hash é a organização eficiente de dados e a otimização de operações, como a busca e a inserção. Cada uma dessas estruturas abordam essas finalidades de maneiras diferentes, dependendo das suas características específicas.

O principal objetivo deste trabalho é realizar uma análise dos algoritmos de busca utilizados nas árvores binárias, árvores AVL e tabelas-hash, com foco especial nos tempos de execução de cada um. Além disso, serão comparados os diferentes cenários em que essas estruturas de dados são mais eficientes, permitindo uma compreensão mais aprofundada de suas características.

1.1 Metodologia

Para o desenvolvimento deste trabalho, foi necessário realizar um estudo aprofundado dos algoritmos e das estruturas de dados apresentados. Com base nesse estudo, foram desenvolvidos códigos equivalentes na linguagem C.

Para obter os dados de tempo de execução, os códigos dos algoritmos foram executados utilizando um script especialmente criado pelo professor João Paulo. Esse script registra os tempos de execução de cada algoritmo para diferentes quantidades de inserções nas estruturas abordadas.

A fim de visualizar e comparar o desempenho dos algoritmos em relação ao tempo, foram utilizados gráficos gerados pela aplicação Gnuplot. Esses gráficos mostram como o tempo de execução varia em diferentes tamanhos das estruturas de dados, proporcionando uma análise mais clara do procedimento de busca.

Os códigos desenvolvidos para gerar os resultados deste trabalho passaram por testes iniciais na plataforma online Replit. Em seguida, foram executados no Windows Subsystem for Linux para a obtenção dos dados necessários.

1.2 Organização do trabalho

O trabalho está organizado em um capítulo de desenvolvimento dividido em subcapítulos. Cada um dos subcapítulos será nomeado de acordo com a estrutura de dados analisada. Os subcapítulos contam com seções que abordam o algoritmo, o código em C, a análise analítica, a representação assintótica e a amostragem do gráfico do tempo de execução da busca em função do tamanho e organização das estruturas.

2. Desenvolvimento

2.1 Árvore Binária

A árvore binária é uma estrutura de dados hierárquica e não linear composta por nós interligados de forma a criar uma estrutura em forma de árvore. Cada nó pode ter até dois nós filhos, conhecidos como filho esquerdo e filho direito. A árvore binária é caracterizada por seguir uma regra específica de ordenação, onde o nó filho esquerdo é sempre menor que o nó pai, e o nó filho direito é sempre maior que o nó pai.

Na Figura 2.1, temos o exemplo de uma árvore binária.

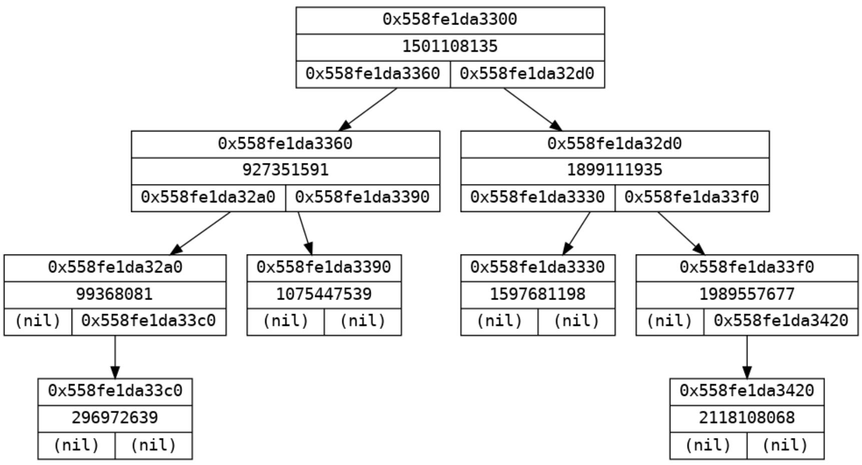

Figura 2.1: Exemplo de árvore binária

2.1.1 Algoritmo base da Árvore Binária

Algoritmo 2.1 (Binary Tree)**.** Árvore binária

algorithm insert($root, node$)
1 if $root = nil$ then
2   $root \leftarrow node$
  else
3   if $root.value < node.value$ then
4   insert($root.right, node$)
    else
5   insert($root.left, node$)

algorithm search($root, value$)

if $root \neq nil$ then
    if $root.value = value$ then
        return $root$
    if $root.value < value$ then
        return search$(root.right, value)$
    return search$(root.left, value)$
return nil $\square$

O algoritmo de busca em uma árvore binária é um processo recursivo que compara a chave de busca com a chave do nó atual e, em seguida, decide em qual direção percorrer a árvore para continuar a busca. O algoritmo de inserção também é baseado em um processo recursivo, onde a posição correta para inserir um novo nó é determinada com base na comparação das chaves.

2.1.2 Código base em C

Segue nas Figuras 2.2, 2.3, 2.4 o código base em C equivalente do Algoritmo 2.1.

```c
struct tree_node {
    int value;
    struct tree_node *left_child;
    struct tree_node *right_child;
};
```

Figura 2.2: Exemplo de apresentação de código em C da estrutura da árvore binária

```c
void insert(struct tree_node **root, int value) {
    if (*root == NULL) {
        *root = malloc(sizeof(struct tree_node));
        (*root)->value = value;
        (*root)->left_child = NULL;
        (*root)->right_child = NULL;
    } else {
        if ((*root)->value < value) {
            insert(&(*root)->right_child, value);
        } else {
            insert(&(*root)->left_child, value);
        }
    }
}
```

Figura 2.3: Exemplo de apresentação de código em C da inserção em uma árvore binária

2.1.3 Análise de casos da Árvore Binária

Em relação ao tempo de execução da busca em uma árvore binária, este está relacionado ao número de nós da árvore e como eles estão inseridos.

```c
struct tree_node *search(struct tree_node *root, int value) {
    if (root != NULL) {
        if (root->value == value) {
            return root;
        }

        if (root->value < value) {
            return search(root->right_child, value);
        }

        return search(root->left_child, value);
    }

    return NULL;
}
```

Figura 2.4: Exemplo de apresentação de código em C da busca em uma árvore binária

Melhor caso

Em termos de busca, o melhor caso ocorre quando o valor buscado está na raiz da árvore. Portanto, o tempo de execução será constante, uma vez que a raiz é o primeiro nó a ser consultado.

Caso médio

O caso médio da árvore binária ocorre quando a árvore não está perfeitamente balanceada. No caso de uma árvore binária não balanceada, o caso médio pode variar consideravelmente dependendo da distribuição dos dados e da estrutura da árvore. Como a árvore não é balanceada, sua altura pode ser significativamente maior do que em uma árvore balanceada, o que pode afetar o desempenho das operações. Segue na Figura 2.5 um exemplo de árvore binária não balanceada.

Em termos de busca, uma árvore binária não balanceada pode ter um caso médio com tempo de execução aproximado entre $O(\log n)$ e $O(n)$, onde n é o número de nós na árvore. Se a árvore estiver estruturada de forma que se aproxima de uma árvore balanceada, o tempo de execução da busca será perto do logarítmico ($O(\log n)$). Entretanto, se os nós estiverem extremamente encadeados à direita ou à esquerda, o tempo de execução será próximo do linear ($O(n)$), ou seja, o pior caso, como veremos a seguir.

Pior caso

O pior caso da busca em uma árvore binária ocorre sempre que a árvore está extremamente desbalanceada, ou seja, quando a árvore funciona como uma lista encadeada unidirecional, seja à direita ou à esquerda, e o valor buscado não está presente na estrutura. Nesse caso, o algoritmo terá que percorrer todos os nós da árvore, a fim de encontrar o valor passado para a função de busca. Por isso, seu tempo de execução é linear ($O(n)$).

Na Figura 2.6, podemos ver uma árvore binária totalmente desbalanceada, que se assemelha a uma lista encadeada.

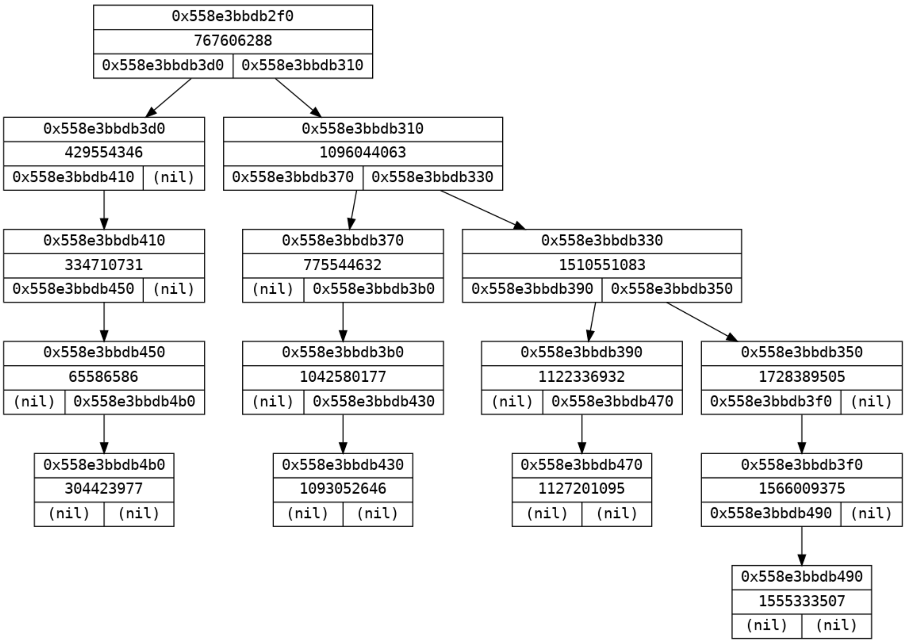

Figura 2.5: Exemplo de árvore binária não balanceada

2.1.4 Calculando o tempo de execução da busca

Pior caso

No pior caso, será necessário realizar uma busca por todos os nós da árvore. Temos então:

$$\text{Caso base} = T_w(0) = c_1 + c_7 = b$$

$$\mathrm{T}_w(\mathrm{n}) = c_1 + c_2 + c_{45} + c_6 + 2 \cdot T_w\left(\frac{n-1}{2}\right)$$

$$\mathrm{T}_w(\mathrm{n}) = a + 2 \cdot T_w\left(\frac{n-1}{2}\right)$$

$$\mathrm{T}_w\left(\frac{n-1}{2}\right) = a + 2 \cdot T_w\left(\frac{\left(\frac{n-1}{2}\right) - 1}{2}\right)$$

$$\mathrm{T}_w\left(\frac{n-1}{2}\right) = a + 2 \cdot T_w\left(\frac{n-3}{4}\right)$$

Substituindo em $T_w(n)$:

$$\mathrm{T}_w(\mathrm{n}) = 3 \cdot a + 4 \cdot T_w\left(\frac{n-3}{4}\right)$$

$$\mathrm{T}_w\left(\frac{n-3}{4}\right) = a + 2 \cdot T_w\left(\frac{\left(\frac{n-3}{4}\right) - 1}{2}\right)$$

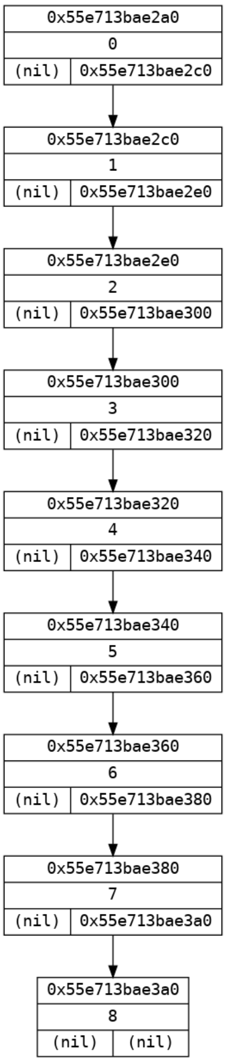

Figura 2.6: Exemplo de árvore binária completamente desbalanceada

$$\mathrm{T}_w\left(\frac{n-3}{4}\right) = a + 2 \cdot T_w\left(\frac{n-7}{8}\right)$$

Substituindo em $T_w(n)$:

$$\mathrm{T}_w(\mathrm{n}) = 7 \cdot a + 8 \cdot T_w\left(\frac{n-7}{8}\right)$$

$$\mathrm{T}_w\left(\frac{n-7}{8}\right) = a + 2 \cdot T_w\left(\frac{\left(\frac{n-7}{8}\right)-1}{2}\right)$$

$$\mathrm{T}_w\left(\frac{n-7}{8}\right) = a + 2 \cdot T_w\left(\frac{n-15}{16}\right)$$

Substituindo em $T_w(n)$:

$$\mathrm{T}_w(\mathrm{n}) = 15 \cdot a + 16 \cdot T_w\left(\frac{n-15}{16}\right)$$

Podemos perceber o padrão:

$$\mathrm{T}_w(\mathrm{n}) = x \cdot a + (x+1) \cdot T_w\left(\frac{n-x}{x+1}\right)$$

Sabemos, pelo caso base:

$$\frac{n-x}{x+1} = 0$$

$$n - x = 0$$

$$n = x$$

Substituindo em $T_w(n)$:

$$\mathrm{T}_w(\mathrm{n}) = n \cdot a + (n+1) \cdot T_w\left(\frac{n-n}{n+1}\right)$$

$$\mathrm{T}_w(\mathrm{n}) = n \cdot a + (n+1) \cdot T_w\left(\frac{0}{n+1}\right)$$

$$\mathrm{T}_w(\mathrm{n}) = n \cdot a + (n+1) \cdot T_w\left(0\right)$$

$$\mathrm{T}_w(\mathrm{n}) = n \cdot a + (n+1) \cdot b$$

$$\mathrm{T}_w(\mathrm{n}) = n \cdot (a+b) + b$$

Logo, comprovamos que no pior caso, a busca em uma árvore binária tem tempo de execução linear.

2.1.5 Análise Assintótica

Melhor caso	$O(1)$
Caso médio	Entre $O(\log n)$ e $O(n)$
Pior caso	$O(n)$

Tabela 2.1: Análise assintótica da busca em uma árvore binária

2.1.6 Gráficos

Melhor caso

A Figura 2.7 contém o gráfico do melhor caso da busca em uma árvore binária.

Ignorando um pouco os erros que podem ocorrer na prática, podemos observar que os tempos de execução se mantêm próximos (quase constantes) na maior parte do gráfico.

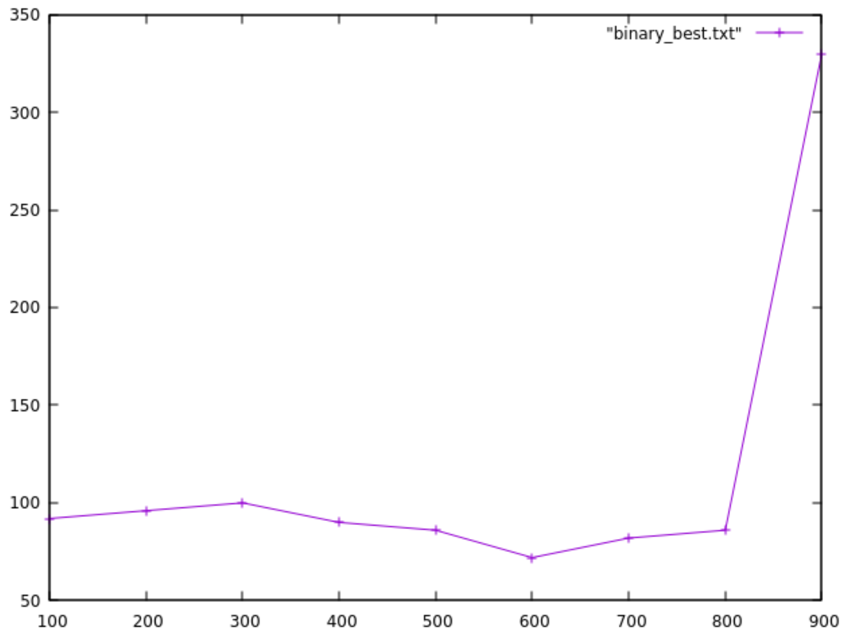

Figura 2.7: Gráfico de desempenho do melhor caso da busca em uma árvore binária

Caso médio

Na Figura 2.8, está o gráfico obtido do caso médio da busca em uma árvore binária. Podemos observar que o gráfico não é uniforme, devido a variação do tempo de execução da busca para árvores não balanceadas.

Pior caso

Na Figura 2.9, podemos visualizar o gráfico do pior caso da busca em uma árvore binária. Nesse caso, como explicado anteriormente, a árvore está extremamente desbalanceada, assumindo uma estrutura semelhante a uma lista encadeada. Essa estrutura de árvore desbalanceada resulta em um tempo de execução linear em relação à busca.

Comparação entre o melhor caso, caso médio e pior caso

Segue na Figura 2.10 um gráfico comparando os três casos já justificados. Podemos perceber a significativa diferença entre os tempos de execução.

2.2 Árvore AVL

Uma árvore AVL é uma estrutura de dados do tipo árvore binária balanceada. O termo “AVL” é derivado dos sobrenomes de seus inventores, Adelson-Velsky e Landis. A principal característica de uma árvore AVL é que ela mantém automaticamente seu fator de balanceamento, garantindo que a diferença entre as alturas das subárvores esquerda e direita de cada nó seja no máximo 1, evitando que um dos lados fique mais “pesado” do que o outro.

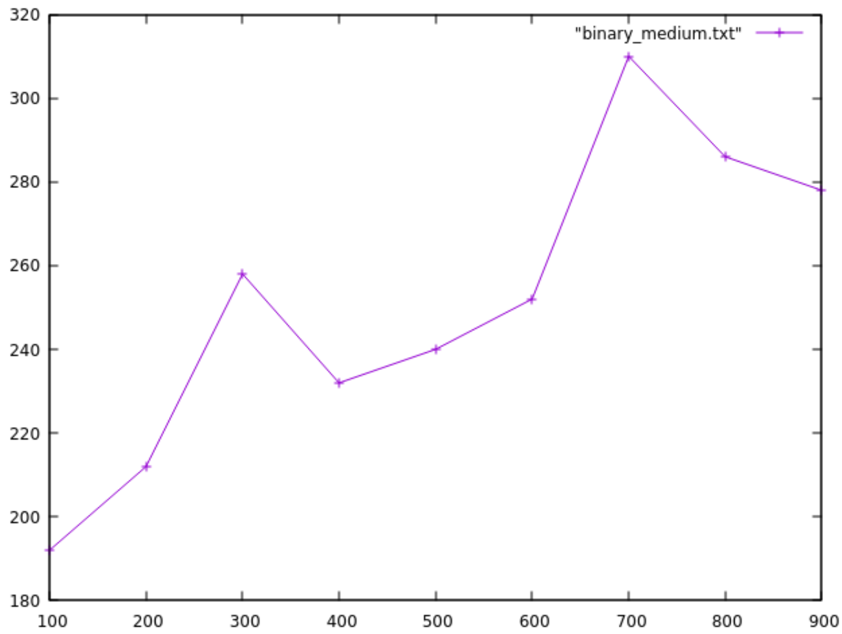

Figura 2.8: Gráfico de desempenho do caso médio da busca em uma árvore binária

Durante a inserção, sempre que um valor é adicionado à árvore, verifica-se se é necessário realizar o balanceamento da estrutura para garantir que a árvore binária esteja sempre balanceada. Isso significa que a árvore AVL favorece a busca, mantendo-a sempre eficiente. Portanto, a árvore AVL é ideal para casos de busca, proporcionando o melhor desempenho possível nesse aspecto.

Na Figura 2.11, temos o exemplo de uma árvore AVL.

2.2.1 Algoritmo base da Árvore AVL

Algoritmo 2.2 (AVL Tree)**.** Árvore AVL

```
algorithm insert(root, node)
    if root = nil then
        root ← node
        balance(node)
    else
        node.parent ← root
        if root.value < node.value then
            insert(root.right, node)
        else
            insert(root.left, node)
algorithm search(root, value)
    if root ≠ nil then
        if root.value = value then
            return r
        if root.value < value then
            return search(root.right, value)
```

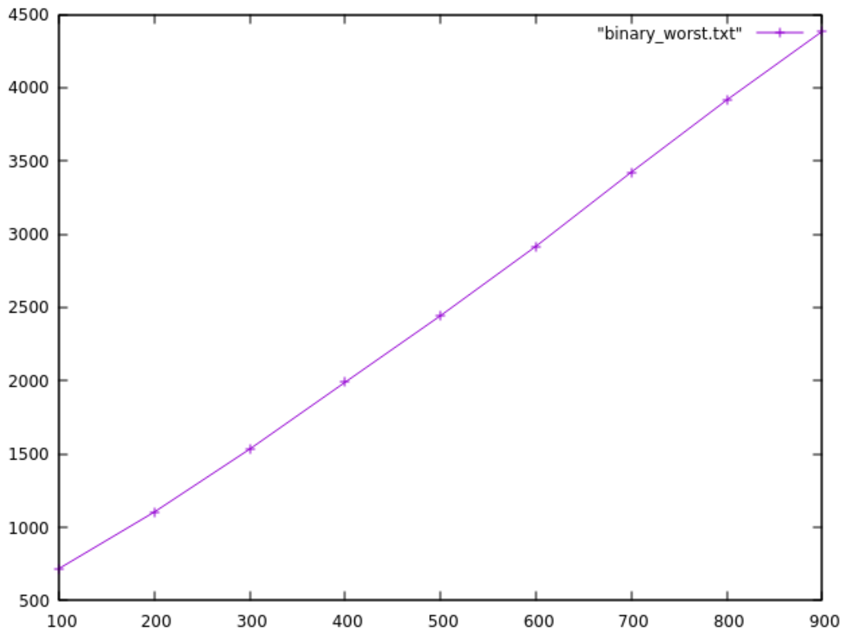

Figura 2.9: Gráfico de desempenho do pior caso da busca em uma árvore binária

```
        return search(root.left, value)
    return nil
algorithm balance(node)
    node.parent ← root
    while node ≠ nil do
        if cdiff(node) > 1 then
            c ← case(node)
            if c = 1 then
                rd(node)
            if c = 2 then
                re(node)
            if c = 3 then
                re(node.left)
                rd(node)
            if c = 4 then
                rd(node.right)
                re(node)
        node ← node.parent
```

□

2.2.2 Código base em C

Segue nas Figuras 2.12, 2.13, 2.14 o código base em C equivalente do algoritmo da árvore AVL. Observa-se que há uma diferença entre este código e o Algoritmo 2.2, pois foi optado por usar uma estratégia diferente no código, atualizando as alturas dos nós de forma recursiva e alocando uma nova estrutura que será retornada ao final da inserção, assim não sendo necessário guardar o “parent” do “node”.

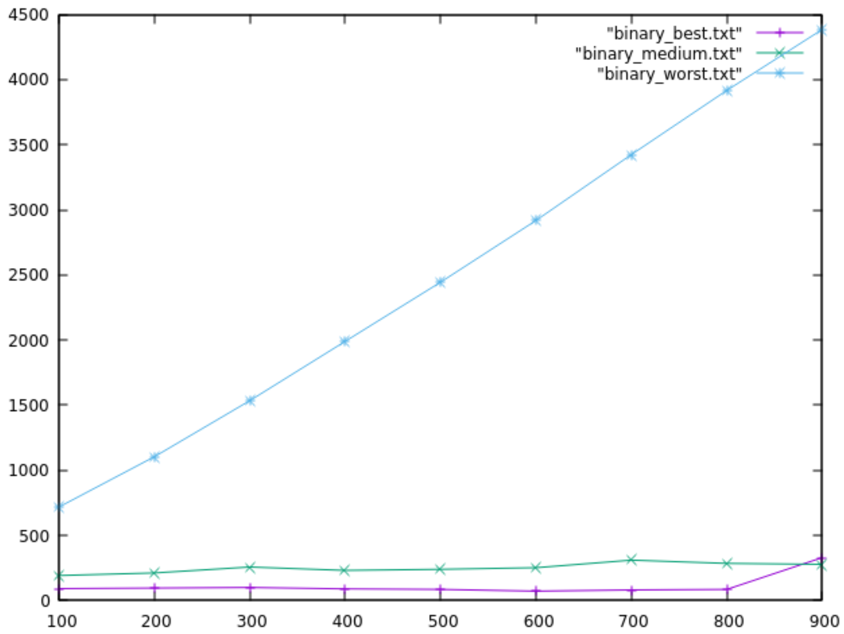

Figura 2.10: Gráfico de comparação entre os três casos da busca em uma árvore binária

2.2.3 Análise de casos da Árvore AVL

Melhor caso

O melhor caso da busca em uma árvore AVL é exatamente o mesmo da busca em uma árvore binária, ou seja, constante, pois em ambos estamos buscando por um valor que está na raiz das estruturas.

Caso médio

Diferentemente da árvore binária comum, que insere os valores tomando como regra apenas as propriedades de esquerda e direita da árvore, a árvore AVL adiciona os valores ao mesmo tempo em que mantém a árvore balanceada. Dessa forma, ao final da inserção, sempre teremos como resultado uma árvore binária balanceada.

Como a árvore binária estará sempre balanceada, percorreremos metade dos nós menos um. O menos um é a própria raiz (primeiro valor da árvore). Esta operação resultará em um tempo de execução logarítmico.

2.2.4 Calculando o tempo de execução da busca

A árvore está balanceada, logo percorreremos metade dos nós menos um.

$$\text{Caso base} = T(0) = c_1 + c_7 = b$$

$$\mathrm{T(n)} = c_1 + c_2 + c_{45} + T\left(\frac{n-1}{2}\right)$$

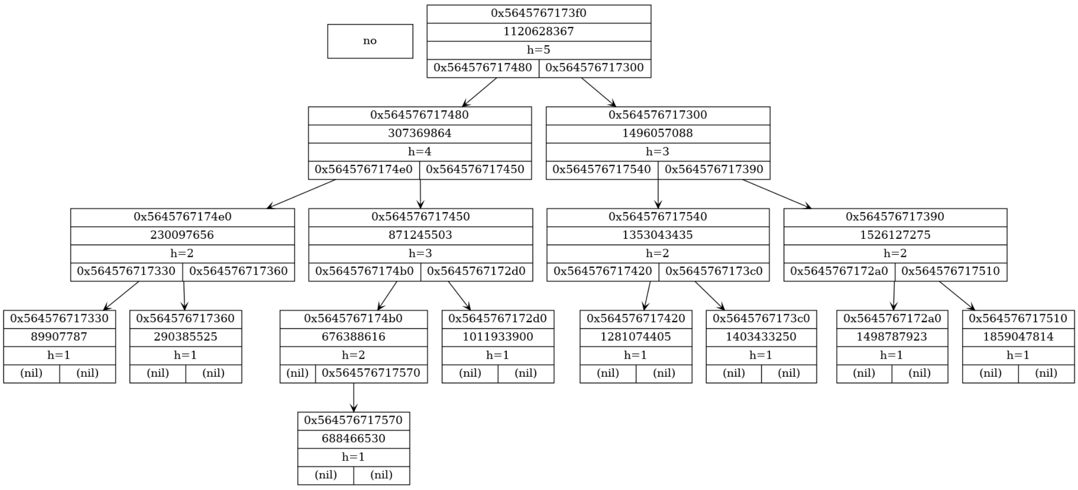

Figura 2.11: Exemplo de árvore AVL

```c
struct tree_node {
    int value;
    struct tree_node* left_child;
    struct tree_node* right_child;
    unsigned int height;
};
```

Figura 2.12: Exemplo de apresentação de código em C da estrutura da árvore AVL

$$\mathrm{T(n)} = a + T\left(\frac{n-1}{2}\right)$$

Como a árvore está balanceada, percorreremos apenas metade dos nós menos um, que é a própria raiz (primeiro valor da árvore).

Vamos resolver a recursão:

$$\mathrm{T}\left(\frac{n-1}{2}\right) = a + T\left(\frac{\left(\frac{n-1}{2}\right)-1}{2}\right)$$

$$\mathrm{T}\left(\frac{n-1}{2}\right) = a + T\left(\frac{n-3}{4}\right)$$

Substituindo em $T(n)$:

$$\mathrm{T(n)} = 2 \cdot a + T\left(\frac{n-3}{4}\right)$$

$$\mathrm{T}\left(\frac{n-3}{4}\right) = a + T\left(\frac{\left(\frac{n-3}{4}\right)-1}{2}\right)$$

$$\mathrm{T}\left(\frac{n-3}{4}\right) = a + T\left(\frac{n-7}{8}\right)$$

Substituindo em $T(n)$:

```c
void update_height(struct tree_node* node) {
    if (node == NULL) {
        return;
    }
    int left_height = get_height(node->left_child);
    int right_height = get_height(node->right_child);
    node->height = (left_height > right_height ?
    left_height : right_height) + 1;
}
```

Figura 2.13: Exemplo de apresentação de código em C da função responsável por atualizar a altura dos nós

$$\mathrm{T(n)} = 3 \cdot a + T\left(\frac{n-7}{8}\right)$$

$$\mathrm{T}\left(\frac{n-3}{4}\right) = a + T\left(\frac{\left(\frac{n-3}{4}\right) - 1}{2}\right)$$

$$\mathrm{T}\left(\frac{n-7}{8}\right) = a + T\left(\frac{\left(\frac{n-7}{8}\right) - 1}{2}\right)$$

$$\mathrm{T}\left(\frac{n-7}{8}\right) = a + T\left(\frac{n-15}{16}\right)$$

304 Substituindo em $T(n)$:

$$\mathrm{T(n)} = 4 \cdot a + T\left(\frac{n-15}{16}\right)$$

305 Podemos perceber o padrão:

$$\mathrm{T(n)} = x \cdot a + T_b\left(\frac{n-(2^x-1)}{2^x}\right)$$

306 Sabemos, pelo caso base:

$$\left(\frac{n-(2^x-1)}{2^x}\right) = 0$$

307 Simplificando:

$$2^x \cdot \left(\frac{n-(2^x-1)}{2^x}\right) = 0 \cdot 2^x$$

$$n - (2^x - 1) = 0$$

$$n - 2^x + 1 = 0$$

$$n + 1 = 2^x$$

```c
struct tree_node* insert(struct tree_node* root, int value) {
    if (root == NULL) {
        root = malloc(sizeof(struct tree_node));
        root->value = value;
        root->left_child = NULL;
        root->right_child = NULL;
        root->height = 1;
    } else if (value < root->value) {
        root->left_child = insert(root->left_child, value);
    } else if (value > root->value) {
        root->right_child = insert(root->right_child, value);
    } else {
        return root;
    }

    update_height(root);
    int difference = get_difference(root);

    if (difference > 1 && value < root->left_child->value) {
        return rotate_right(root);
    }
    if (difference < -1 && value > root->right_child->value) {
        return rotate_left(root);
    }
    if (difference > 1 && value > root->left_child->value) {
        root->left_child = rotate_left(root->left_child);
        return rotate_right(root);
    }
    if (difference < -1 && value < root->right_child->value) {
        root->right_child = rotate_right(root->right_child);
        return rotate_left(root);
    }

    return root;
}
```

Figura 2.14: Exemplo de apresentação de código em C da inserção em uma árvore AVL

$$\log_2 n + 1 = \log_2 2^x$$

$$x = \log_2 n + 1$$

Substituindo em $T(n)$:

$$\mathrm{T(n)} = a \cdot \log_2(n+1) + T\left(\frac{n - (2^{\log_2 n + 1} - 1)}{2^x}\right)$$

$$\mathrm{T(n)} = a \cdot \log_2(n+1) + T\left(0\right)$$

$$\mathrm{T(n)} = a \cdot \log_2(n+1) + b$$

Logo, comprovamos que no caso médio, a busca na árvore AVL tem tempo de execução logarítmico.

2.2.5 Análise Assintótica

Melhor caso	$O(1)$
Caso médio	$O(\log n)$

Tabela 2.2: Análise assintótica da busca em uma árvore AVL

2.2.6 Gráficos

Melhor caso

Como o melhor caso da busca em uma árvore binária e em uma AVL é o mesmo, os gráficos também serão iguais, ou seja, interpolando valores próximos de constantes. Segue na Figura 2.15.

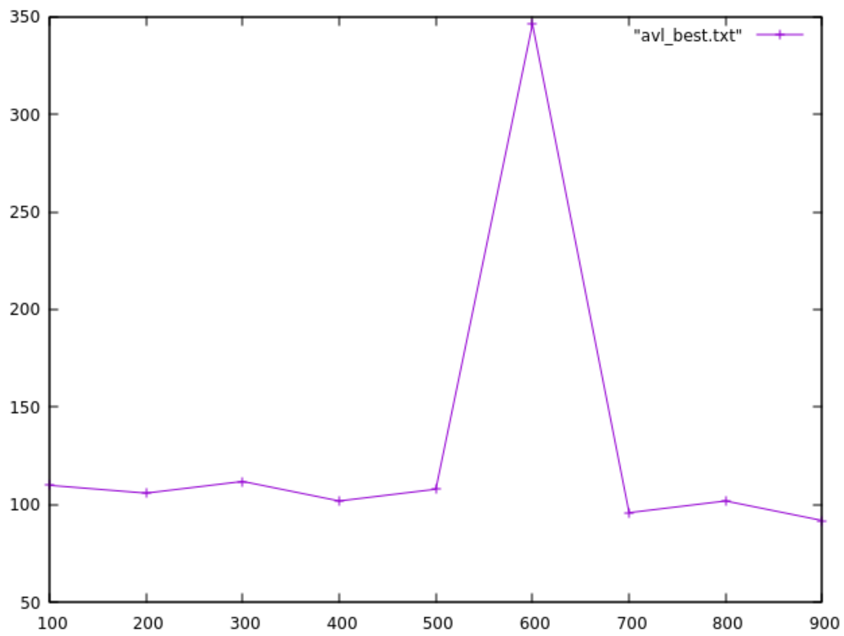

Figura 2.15: Gráfico de desempenho do melhor caso da busca em uma árvore AVL

Caso médio

O gráfico na Figura 2.16 retrata o resultado obtido na execução do caso médio da busca em uma árvore AVL. O gráfico se aproxima da curva característica da função logarítmica ($\log(n)$).

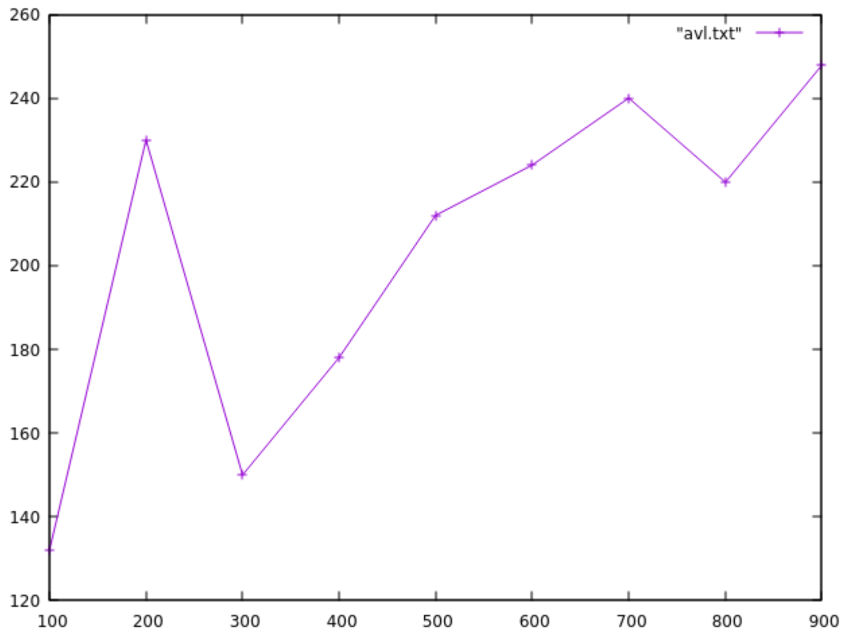

Figura 2.16: Gráfico de desempenho do caso médio da busca em uma árvore AVL

Comparação entre os casos da árvore AVL

Segue na Figura 2.17 o gráfico comparativo entre os casos da busca em uma árvore AVL.

2.3 Tabela-Hash

A tabela-hash, também conhecida como tabela de dispersão é uma estrutura de dados que facilita o armazenamento, acesso e busca de dados de forma eficiente. O funcionamento desta estrutura é dependente de uma função hash, ou função de dispersão, que definirá a posição do elemento na tabela.

Uma tabela-hash geralmente consiste em elementos associados a uma posição do vetor de ponteiros que representa a tabela, através da função hash. Quando um valor precisa ser associado na tabela, a função hash é aplicada a esse valor para determinar sua posição de armazenamento. Caso ocorra uma colisão, ou seja, dois valores diferentes mapeados para a mesma posição na tabela, existem várias técnicas para resolver esse problema, como o uso de encadeamento de estruturas.

Segue na Figura 2.18 um exemplo de tabela-hash.

Neste trabalho, optou-se por desenvolver uma tabela-hash com uma função de dispersão $x \mod m$, onde x representa o valor que queremos inserir e m é o tamanho do vetor das posições da tabela. Quando $\frac{m}{n} > 1$, utilizamos a estratégia do *rehashing*, uma função responsável por aumentar o tamanho da tabela e reorganizar os dados para evitar que as posições tenham muitos valores associados. Essa estratégia é importante para evitar o pior caso em termos de eficiência de busca, como veremos mais adiante.

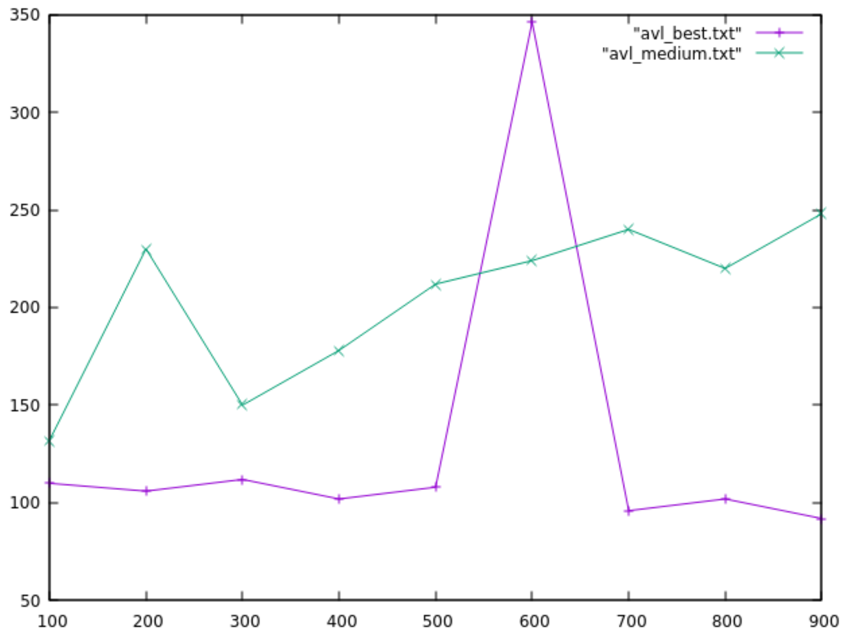

Figura 2.17: Gráfico de desempenho do caso médio da busca em uma árvore AVL

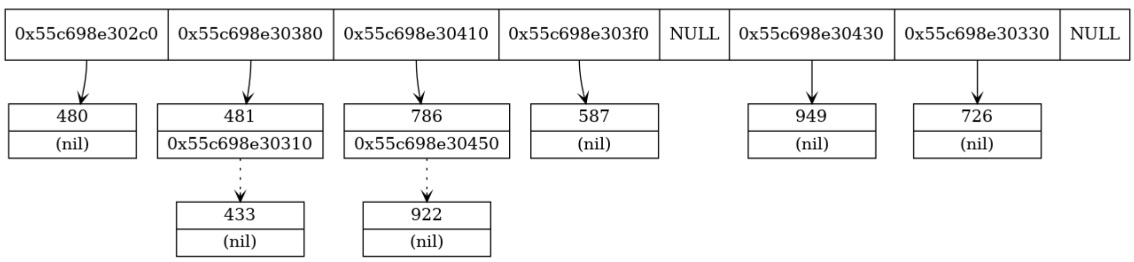

Figura 2.18: Exemplo de tabela-hash

2.3.1 Código base em C

Nas Figuras 2.19, 2.20, 2.21, 2.22 temos um exemplo de código base da tabela-hash.

2.3.2 Análise de casos da Tabela-Hash

O tempo de execução da busca em uma tabela-hash está relacionado ao número de valores associados ao vetor de posições da tabela. Caso ocorra colisões na inserção dos valores, uma lista encadeada é gerada até que ocorra o *rehashing*.

Melhor caso

O melhor caso da busca em uma tabela-hash ocorre quando não há colisões. Isso significa que cada posição da tabela aponta para um único nó, permitindo que a busca seja executada em tempo constante $O(1)$.

Em uma tabela hash sem colisões, quando você deseja procurar um elemento específico, o algoritmo de hash calcula a posição exata na tabela onde esse elemento está armazenado. A busca é então realizada diretamente nessa posição e o elemento é encontrado

```c
typedef struct Node {
    int value;
    struct Node* next;
} Node;

typedef struct {
    int m;
    int n;
    Node** nodes;
} HashTable;
```

Figura 2.19: Exemplo de apresentação de código em C da estrutura da tabela-hash

```c
int hash(int value, int m) {
    return value % m;
}
```

Figura 2.20: Exemplo de apresentação de código em C da função hash

imediatamente, independentemente do tamanho da tabela.

Na Figura 2.23 temos um exemplo de uma tabela-hash que gerará o melhor caso em relação a busca.

Caso médio

Em relação à busca, o caso médio em uma tabela hash ocorre quando os valores estão distribuídos de forma relativamente uniforme na tabela, mas não de maneira exclusiva para cada posição, como no melhor caso. Isso pode resultar em colisões e levar à formação de pequenas listas encadeadas.

Portanto, o tempo de execução da busca, neste caso, irá variar entre constante ($O(1)$) e linear ($O(n)$).

Segue na Figura 2.24 um exemplo de tabela-hash que resultará no caso médio em relação a busca.

Pior caso

O pior caso da busca em uma tabela-hash, ocorre quando os nós estão armazenados em uma única posição, ou seja, todos os valores entram em colisão. Assim, a busca terá que percorrer uma lista encadeada, resultando em um tempo de execução linear ($O(n)$).

Segue na Figura 2.25 um exemplo de tabela-hash que resultará no pior caso em relação a busca.

2.3.3 Calculo do tempo de execução esperado da busca

Para calcular o tempo de execução esperado da busca em uma tabela-hash, precisamos somar todas possibilidades, incluindo a do pior caso.

Possibilidade de cair no melhor caso:

$$\mathrm{P}_b = \frac{1}{n} \cdot c + \frac{1}{n} \cdot c + \frac{1}{n} \cdot c + \dots$$

```c
void insert(HashTable* table, int value) {
    int key = hash(value, table->m);

    if ((float)table->n / table->m >= 1.0) {
        rehash(table, value);
    } else {
        int index = hash(key, table->m);
        Node* new_node = create_node(value);

        if (table->nodes[index] == NULL) {
            table->nodes[index] = new_node;
        } else {
            Node* current_node = table->nodes[index];

            while (current_node->next != NULL) {
                current_node = current_node->next;
            }
            current_node->next = new_node;
        }

        table->n++;
    }
}
```

Figura 2.21: Exemplo de apresentação de código em C da inserção em uma tabela-hash

O termo “c” é uma constante, referente ao tempo de execução da busca em uma posição da tabela que aponta para *nulo* ou somente para um nó.

Supondo que todos os nós estão organizados em uma só posição do vetor da tabela, temos:

$$\mathrm{T}_a(n) = \frac{1}{n} \cdot c + \frac{1}{n} \cdot c + \frac{1}{n} \cdot c + \ldots + \frac{1}{n} \cdot c \cdot n$$

O “n” é a quantidade de nós encadeados.

Podemos reescrever esta sequência da seguinte forma:

$$\mathrm{T}_a(n) = (n-1) \cdot \frac{1}{n} \cdot c + \ldots + \frac{1}{n} \cdot c \cdot n$$

Reescrevemos como $n-1$, pois todas as posições menos a última terão um tempo de execução de busca constante.

$$\mathrm{T}_a(n) = \frac{1}{n} \cdot [n \cdot c - c + n \cdot c] = \frac{1}{n} [2 \cdot n \cdot c]$$

$$\mathrm{T}_a(n) = \frac{c}{n} \cdot (2 \cdot n - 1) = c \cdot \left(2 - \frac{1}{n}\right)$$

Observando o resultado, é possível perceber que a função tende a ser constante, mesmo aumentando o tamanho de “n”.

Logo,

```c
int search(HashTable* table, int value) {
    int index = hash(value, table->m);
    Node* current_node = table->nodes[index];

    while (current_node != NULL) {
        if (current_node->value == value) {
            return 1;
        }
        current_node = current_node->next;
    }

    return 0;
}
```

Figura 2.22: Exemplo de apresentação de código em C da busca em uma tabela-hash

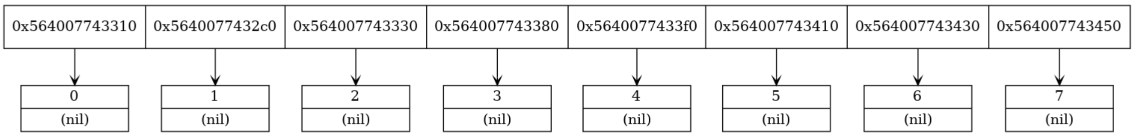

Figura 2.23: Exemplo de tabela-hash que ocasiona o melhor caso da busca

$$T_a(n) \in \Theta(1)$$

Assim, comprovamos que o tempo de execução esperado da busca em uma tabela-hash tende a ser constante.

2.3.4 Análise Assintótica

Melhor caso	$O(1)$
Caso médio	Entre $O(1)$ e $O(n)$
Pior caso	$O(n)$

Tabela 2.3: Análise assintótica da busca na tabela-hash

2.3.5 Gráficos

Melhor caso

Segue na Figura 2.26 o gráfico obtido do melhor caso da busca em uma tabela-hash.

Ignorando um pouco os erros que podem ocorrer na prática, podemos observar que os tempos de execução se mantêm próximos (quase constantes) na maior parte do gráfico.

Caso médio

Na Figura 2.27, temos o gráfico obtido na execução do algoritmo de busca em uma tabela-hash no caso médio. Vale ressaltar que o gráfico para esse caso não é bem definido devido ao fato de que o tempo de execução pode variar consideravelmente.

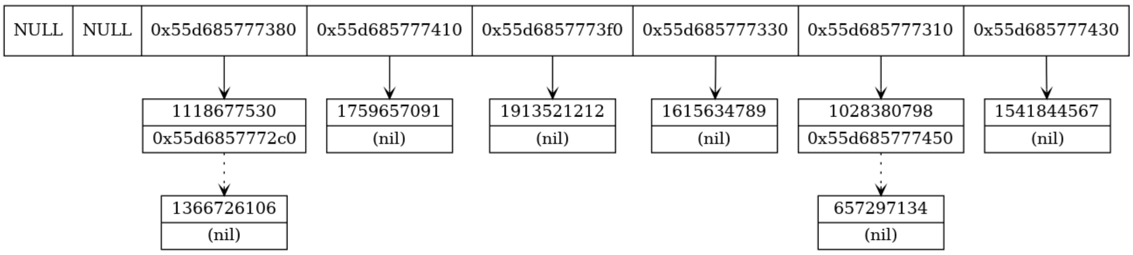

Figura 2.24: Exemplo de tabela-hash que ocasiona o caso médio da busca

Novamente, ignorando um pouco os erros que podem ocorrer na prática, podemos observar que os tempos de execução se mantêm próximos (quase constantes), mas com algumas pequenas partes lineares.

Pior caso

Na Figura 2.28, podemos visualizar o gráfico obtido na execução da busca em uma tabela-hash no pior caso. Conforme mencionado anteriormente, no pior caso, haverá uma posição que apontará para uma lista encadeada, resultando em um tempo de execução linear para a busca.

Todos os casos

Segue na Figura 2.29 um gráfico comparativo entre todos os casos da tabela-hash.

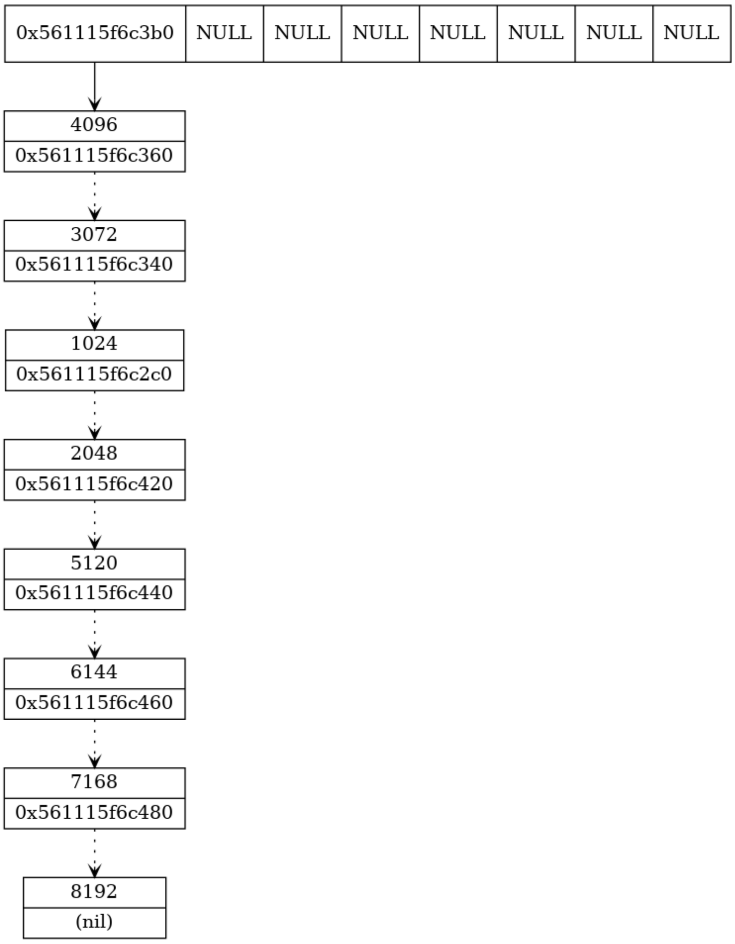

Figura 2.25: Exemplo de tabela-hash que ocasiona o pior caso da busca

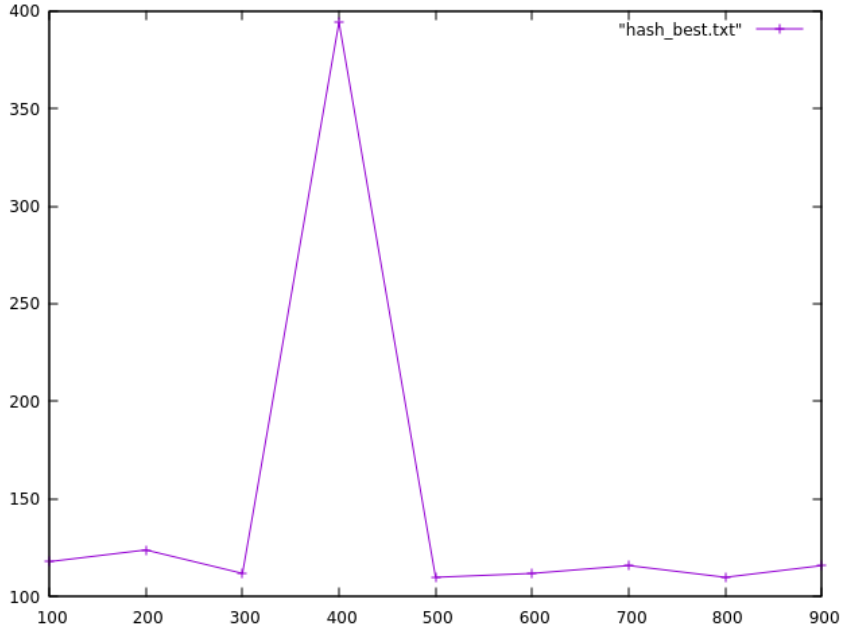

Figura 2.26: Gráfico de desempenho da busca em uma tabela-hash no melhor caso

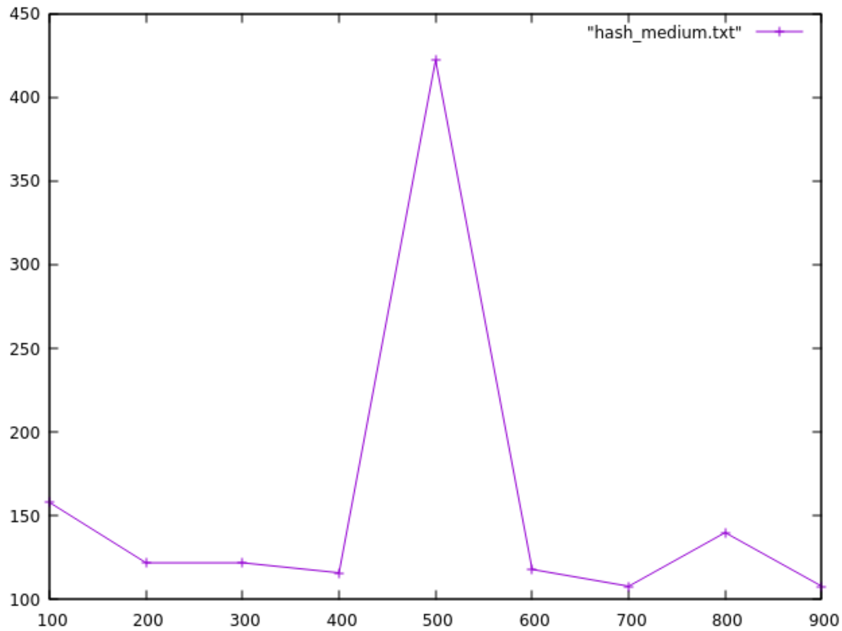

Figura 2.27: Gráfico de desempenho da busca em uma tabela-hash no caso médio

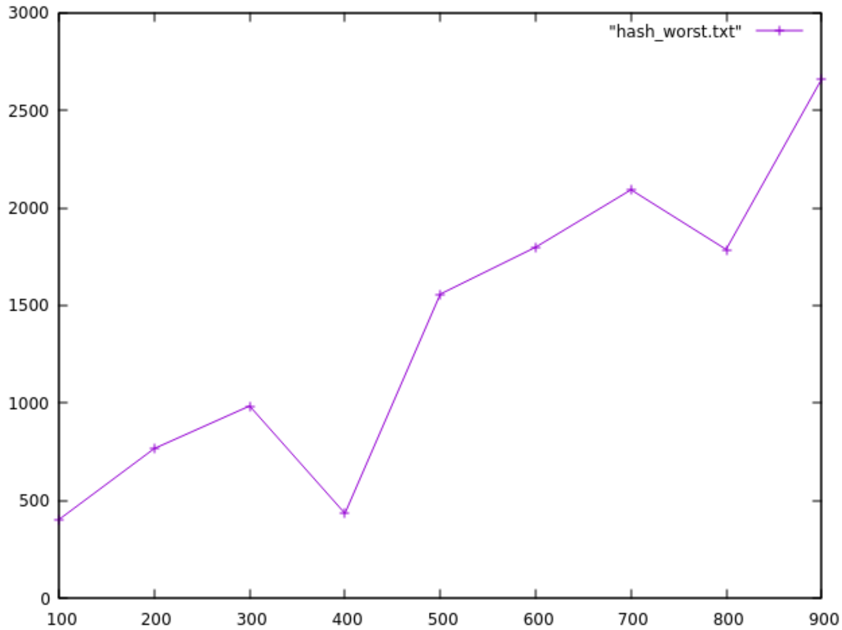

Figura 2.28: Gráfico de desempenho da busca em uma tabela-hash no pior caso

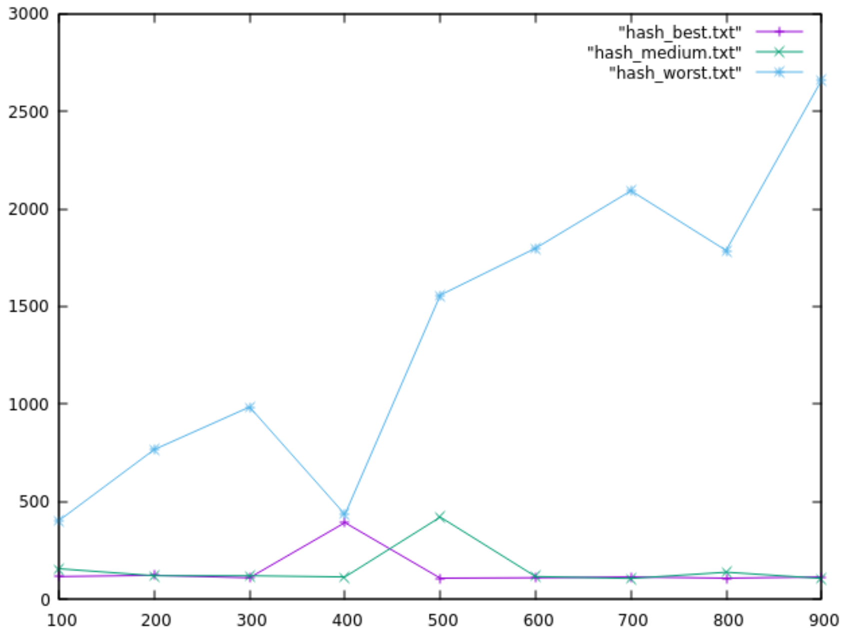

Figura 2.29: Gráfico de comparação de todos os casos da tabela-hash

3. Conclusões

3.1 Resultados

Agora faremos uma análise dos resultados obtidos de cada algoritmo apresentado ao longo do trabalho. Para as comparações, será levado em consideração principalmente os casos médios da busca em cada estrutura de dados estudada.

Comparando as estruturas em árvore, a AVL é uma variação da árvore binária que impõe restrições adicionais para garantir um balanceamento automático. O balanceamento é importante para evitar que a árvore se torne muito desequilibrada e reduza o desempenho das operações. A principal vantagem da árvore AVL em relação à árvore binária é que ela garante um balanceamento de altura, o que resulta em um desempenho mais eficiente para as operações de busca.

Portanto, é mais vantajoso utilizar a estrutura de árvores AVL para a organização dos dados, pois garante um acesso mais rápido, ao contrário da árvore binária comum, onde não temos garantia de que os dados estarão bem distribuídos. Podemos confirmar essa afirmação analisando o gráfico presente no capítulo de árvores AVL.

Caso seja necessário um tempo de busca ainda mais eficiente, temos a estrutura de tabela-hash. Como foi calculado anteriormente neste trabalho, o tempo de execução da busca nessa estrutura de dados tende a ser constante, tornando-a a estrutura em que a busca funciona de maneira mais rápida.

A diferença entre as duas estruturas mais eficientes apresentadas neste trabalho (árvore AVL e tabela-hash) está na complexidade dos casos de busca. Enquanto a busca na árvore AVL terá um tempo de execução de $O(\log n)$, o desempenho da tabela hash depende da organização dos dados. No cenário ideal, se o algoritmo de inserção na tabela de dispersão estiver satisfatório e a função de hash e rehash forem eficientes, será evitado o pior caso. No entanto, se não for o cenário ideal, os dados podem ficar encadeados em listas, resultando em um tempo de execução linear na busca $O(n)$.

3.2 Comparando os tempos de execução

Por fim, vamos visualizar por meio da Tabela 3.1 e da Figura 3.1 comparações entre os tempos de execução obtidos neste trabalho que justificam os resultados apresentados para cada algoritmo.

3.3 Ressalvas

Para a realização deste trabalho, os códigos em C foram testados no Replit e executados para a geração dos gráficos no Windows Subsystem For Linux. Como não foi

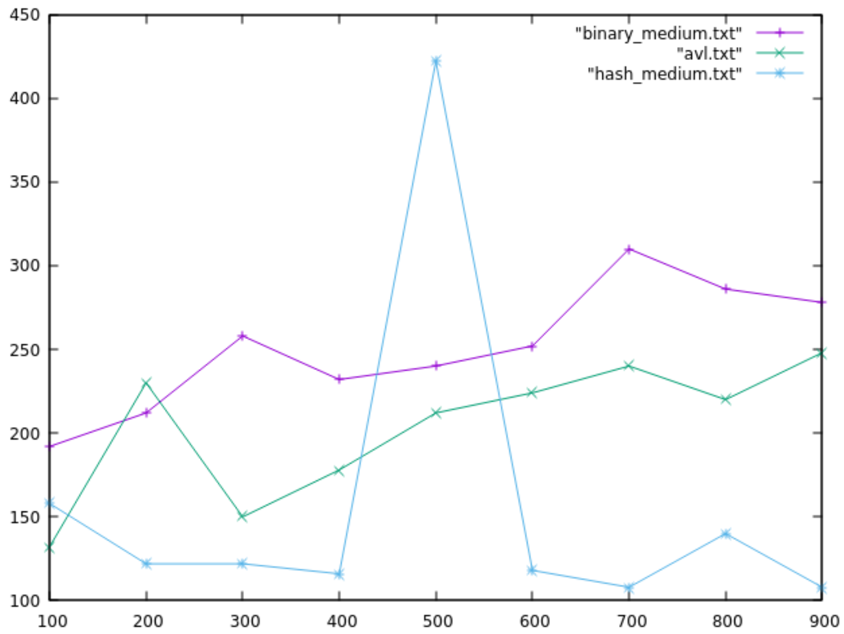

Figura 3.1: Gráfico de comparação dos casos médios das estruturas de dados estudadas

n	Árvore Binária	Árvore AVL	Tabela-Hash
100	192	132	158
200	212	230	122
300	258	150	122
400	232	178	116
500	240	212	422
600	252	224	118
700	310	240	108
800	286	220	140
900	278	248	108

Tabela 3.1: Tempos de execução em nanosegundos de cada algoritmo para diferentes números de valores inseridos

utilizado um sistema Linux nativamente, os tempos obtidos podem parecer destoantes do ideal, caso fossem rodados num sistema Linux nativo. Mesmo assim, os resultados foram surpreendentementes positivos.

3.4 Considerações Finais

Neste trabalho, pudemos explorar algumas das estruturas de dados mais conhecidas. Conseguimos explicar seus devidos funcionamentos, calcular os tempos de execução de cada um em diversos cenários, além de registrar, de maneira gráfica, os tempos de execução dos algoritmos tratados neste documento.